

Activité 3

Fonctions, tracé de courbes

1 Quelques rappels « théoriques »...

1.1 Fonctions et procédures

a Objectif

Dès qu'un programme s'étoffe un peu, il devient peu lisible s'il est écrit « linéairement ». On préfère en général (et c'est ce que nous avons vu dans le chapitre précédent) le décomposer en plusieurs « sous programmes » (ensemble d'instructions) appelés par le programme principal.

Parfois certaines actions doivent être effectuées plusieurs fois... les écrire dans un sous programme est alors un choix judicieux.

Certaines instructions peuvent également être utilisées dans différents programmes ; les inclure dans un sous programme (puis éventuellement dans une bibliothèque) permettra de les avoir facilement à disposition.

b Implémentation

On distingue deux types de « sous programme » :

- les `procédures` qui se « contentent » d'exécuter une série d'instruction sans retourner de valeur ;
- les `fonctions` qui retournent systématiquement une (ou plusieurs) valeurs grâce à l'instruction `return`.

Procédures et fonctions sont :

- définis par leur nom grâce à l'instruction `def maFonction(x,y) :` où *maFonction* est le nom que l'on souhaite donner à la fonction ou à la procédure et *x*, *y*... les paramètres que l'on souhaite transmettre à la fonction (ou à la procédures). Ces paramètres, en nombre quelconques sont introduits entre parenthèses et séparés par des virgules ;
- appelés dans le corps du programme principal par « *maFonction*(2,3) » par exemple ;
- dans le cas d'une fonction, la(les) valeur(s) retournée doivent être stockées dans des variables : « *z* = *maFonction*(2,3) » ; la variable *z* contient alors le résultat du calcul effectué par *maFonction* lorsqu'elle reçoit les valeurs 2 et 3 comme paramètres.

c Pièges et difficultés

La principale difficulté, au début, est de bien comprendre la notion de paramètres à passer à une fonction ou à une procédure. Comme sur votre calculatrice, il ne suffit pas d'écrire *sin*, par exemple,

pour que le sinus du nombre auquel on pense soit calculé!

L'erreur classique, ensuite, consiste à supposer que du moment qu'on appelle une fonction avec les bons paramètres, le résultat du calcul est stocké quelque part... Le calcul est bien effectué mais la valeur de retour est perdue si on ne la stocke pas dans une variable : il faut absolument, dans le cas d'une fonction, avoir une structure du type : `variable_résultat = maFonction(paramètres)`. Abordons, ici, un cas plus délicat ; celui où une fonction retourne plusieurs valeurs.



On ne peut avoir qu'un seul « return » par fonction... les différentes valeurs ne doivent donc pas être retournées par plusieurs « return » de suite.

Par contre, on peut écrire « return valeur1, valeur2 » ; et on récupérera les résultats sous la forme `variable1, variable2 = maFonction(paramètres)`.

Par exemple, la fonction suivante retourne le périmètre et la surface d'un carré.

```
1 def carre(a):
2     p = 4*a
3     s = a*a
4     return p, s
5
6 perimetre, surface = carre(3)
7 print("Périmètre : " + str(perimetre) + " surface : " +
      str(surface))
```

1.2 Application : tracé de courbes

Une des principales applications des fonctions est quand même de pouvoir définir... une fonction ! et tracer sa courbe représentative.

Soit, par exemple, la représentation de la fonction $\sin(k \times x)$ entre 0 et $2 * \pi$.

Le programme « élémentaire » pourrait être le suivant. Il fait appel à la bibliothèque « matplotlib.pyplot » pour le tracé de courbes. Cette bibliothèque est extrêmement complète mais son utilisation élémentaire est très simple.

```
1 from math import sin, pi
2 import matplotlib.pyplot as plt
3
4 X = [] #tableau de valeurs pour les abscisses
5 Y = [] #tableau de valeurs pour les ordonnées
6 dX = pi/100 #incrément des valeurs sur x
7
8 def sinkx(k,x): #on définit la fonction que l'on veut étudier
9     return sin(k*x)
10
11 x = 0 #valeur de x en cours
12 while x <= 2*pi :#on parcourt une boucle tant que x est <= à 2
13     pi (cf chapitre suivant)
14     X.append(x) # on complète le tableau des abscisses
15     Y.append(sinkx(4, x)) # et celui des ordonnées
16     x = x + dX # on n'oublie pas d'incrémenter la valeur de x
17
18 """ cette partie vous est donnée """
19 plt.plot(X, Y)# on a, au minimum, besoin des deux tableaux de
    valeurs X et Y
20 plt.show()
```

On se reportera à l'annexe pour enrichir éventuellement ce graphe.

1.3 Application : fonction récursive

Une fonction (ou un programme) est dite récursive si elle s'appelle elle-même, une ou plusieurs fois. Un exemple très classique est la programmation de la fonction factorielle.

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \times fact(n-1) & \text{sinon} \end{cases}$$

Fonction $fact(n)$

Entrées : un entier n

Sorties : la quantité $n!$

si $n == 0$ **alors**

| retourner 1

sinon

| retourner $n \times fact(n-1)$

fin

```
1 def fact(n) :
2     if n==0 :
3         return 1
4     else :
5         return n*fact(n-1)
```

2

A vous de jouer...

2.1

Blondes

Il n'y en a plus désormais !

2.2

Châtain clair...

.1 Courbe du Blanc-Mange

Dans cet exercice, on cherche à tracer une approximation de la courbe du Blanc-Mange (en référence à un dessert qui ressemble à cette courbe).

On définit les fonctions B_n , avec n un entier par :

$$B_n(x) = \sum_{k=0}^n \frac{1}{2^k} \left| 2^k x - E \left(2^k x + \frac{1}{2} \right) \right|$$

- La fonction E est la fonction partie entière. On la trouve en python sous le nom `floor`, dans la bibliothèque `math`. (`floor(12.95) = 12`)
- Les barres verticales désignent la valeur absolue.



Plus n est grand, plus la courbe de B_n ressemble à la "vraie" courbe du blanc-mange.

1. Créer une fonction python $B(n, x)$ qui renvoie la valeur de $B_n(x)$.
2. Tracer la courbe de B_n sur $[0, 1]$. A tester pour $n = 10, 100, 1000$ par exemple.

.2 Triangle de Pascal Écrire un programme qui affiche le triangle de pascal à n lignes. On pourra d'abord programmer une fonction qui prend en paramètre une liste qui contient une ligne du triangle, et renvoie la ligne suivante.

L'exemple suivant montre une façon de formater le texte avec des colonnes bien alignées.

```

1 liste1 = [2,12,52]
2 liste2 =[152,0, 6]
3 chaine1 = ""
4 chaine2 = ""
5 for n in liste1 :
6     chaine1 = chaine1 + "{:<4d}".format(n)
7 for n in liste2 :
8     chaine2 = chaine2 + "{:<4d}".format(n)
9 print(chaine1)
10 print(chaine2)
```

2.3 Réactions successives (Acte II)

Retour sur les réactions successives : $A \xrightarrow{k_1} B \xrightarrow{k_2} C$.

Partant d'une concentration A_0 en **A**, l'expression analytique des concentrations en fonction du temps est :

$$A(t) = A_0 e^{-k_1 t}; B(t) = \frac{k_1 A_0}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t}); C(t) = A_0 \left(1 + \frac{k_1 e^{-k_2 t} - k_2 e^{-k_1 t}}{k_2 - k_1} \right)$$

1. Définir les fonctions $A(k_1, k_2, t)$, $B(k_1, k_2, t)$ et $C(k_1, k_2, t)$ et tracer pour des valeurs de k_1 et k_2 de votre choix les courbes représentatives.
2. Écrire une fonction $maximum(k_1, k_2)$ qui pour les valeurs de k_1 et k_2 passées en paramètre retourne le temps pour lequel la concentration en **B** passe par un maximum et la valeur de ce maximum de **B**. Faire afficher les valeurs dans les deux cas de figure précédents.

On complique un peu...

3. On prend la valeur $k_1 = 1$. On définit le rapport des constantes de vitesse $r = \frac{k_2}{k_1}$. Tracer la courbe qui donne la concentration maximale atteinte par **B** en fonction du logarithme décimal de r pour $\log_{10}(r)$ variant de -2 à +2.
4. On peut considérer que l'on peut appliquer l'AEQS si la concentration en **B** ne dépasse pas 5% de la concentration initiale A_0 ; quelle est la valeur de r correspondant à cette limite.

2.4 Régression linéaire

On considère un nuage de n points $M_i(x_i, y_i)$ que l'on désire ajuster au mieux par une courbe $y = a \times x + b$. Dans la méthode des moindres carrés, on cherche à minimiser la somme des distances entre les points M_i et la droite.

On note :

- \bar{x} et \bar{y} la moyenne des X et des Y ;
- $\sigma_x = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$ et $\sigma_y = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2}$ les écarts types correspondants ;

- $Cov(X, Y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}) \times (y_i - \bar{y})$ la covariance (moyenne du produit des écarts à la moyenne)

Dans ces conditions :

- le coefficient de corrélation linéaire vaut $r(X, Y) = \frac{Cov(X, Y)}{\sigma_x \sigma_y}$;
 - la pente de la droite de corrélation : $a = \frac{Cov(X, Y)}{(\sigma_x)^2}$
 - l'ordonnée à l'origine de la droite de corrélation vaut $b = \bar{y} - a\bar{x}$
1. Écrire la routine *regrelin*(*X*, *Y*). Les paramètres d'appel de cette fonction sont les deux tableaux *X* et *Y* (de même dimension) ; la valeur retournée : un tuple¹ contenant, dans l'ordre, *a*, *b* et le coefficient de corrélation *r*.
 2. Tester à l'aide des tableaux de valeurs *X* = [1, 2, 3, 4, 5] et *Y* = [3.1, 4.9, 7, 8.9, 11.2]. On trouve, sur Excel *a* = 2,02, *b* = 0,96 et *r*² = 0,99843.
 3. Etude de la réaction de saponification du propanoate d'éthyle. On donne les tableaux de valeur *theta* = [35, 40, 45, 50] et *k* = [0.188, 0.257, 0.354, 0.477]. Le premier correspond aux températures (en °C) ; le second aux constantes de vitesse (en $\text{mol}^{-1} \cdot \text{L} \cdot \text{s}^{-1}$).
 - a. Introduire les tableaux de valeur *theta* et *k* ; construire les tableaux de valeurs *unSurT* et *lnk* contenant, respectivement, $\frac{1}{T(\text{enK})}$ et $\ln(k)$.
 - b. Tracer la courbe en ne matérialisant que les points expérimentaux.
 - c. Faire la régression linéaire à l'aide de votre routine *regrelin*. Afficher la valeur de l'énergie d'activation de la réaction.
 - d. Superposer au graphique précédent la droite de régression.
 4. Écrire une routine *regrelinEntre*(*X*, *Y*, *xmin*, *ymin*) qui effectue une régression linéaire sur les tableaux *X* et *Y* en se limitant aux abscisses comprises entre *xmin* et *ymin*.
 5. Copier l'intégralité du code correspondant aux regressions linéaires dans un fichier nommé, par exemple utilitaire.py que vous placez dans votre dossier de travail. Ecrire from utilitaire import * permet alors d'utiliser ces routines dans n'importe quel programme. Essayez !

1. 3 valeurs séparées par des virgules

3 Quelques pistes...

3.1 Blondes

Le début de l'exercice sur les courbes de répartition est très simple. L'exercice sur la régression linéaire montre comment articuler les différents appels de fonctions pour résoudre un problème posé.

3.2 Châtain clair...

.1 Courbe du Blanc-Mange

C'est surtout un exercice de gestion de parenthèses ! Si on a peur de se tromper dans les priorités, mieux vaut en mettre plus (sans abuser toutefois). Avec spyder, cliquer juste après une parenthèse permet de mettre en surbrillance la parenthèse ouvrante ou fermante correspondante.

Pour $n = 1000$, j'ai pris un pas de 0,001.

.2 Triangle de Pascal

On pourra programmer une fonction qui prend en paramètre une liste qui contient une ligne du triangle, et renvoie la ligne suivante ! Cette approche est peut-être plus simple à programmer qu'utiliser la formule du nombre des combinaisons.

3.3 Réactions successives

1. Au lieu d'avoir k_1 et k_2 paramètres des fonctions, vous pouvez les introduire comme variables globales du système. Pour anticiper un peu sur la question 3, vous pouvez créer une fonction qui génère les 4 tableaux de valeurs nécessaires en passant à cette fonction les paramètres adéquats.
2. Il s'agit de réinvestir :) ce que vous aviez fait la semaine dernière... ne me dites pas que vous ne savez plus faire !
3. Pour éviter le cas de figure $k_1 \approx k_2$ dans l'expression de $B(t)$, je n'ai pas fait de calcul pour $|\log r| < 0,1$.
4. Au début j'avais pris un critère de 1%, il faut alors explorer aller jusqu'à $\log r = 3$.

3.4 Régression linéaire

1. En principe, vous n'avez que 3 fonctions (autre que `regrelin`) à programmer !
2. C'est l'heure de vérité pour votre fonction !
3. Essayez de construire les tableaux `unSurT` et `lnk` à l'aide de listes par compréhension, c'est une façon « élégante » de traiter des tableaux expérimentaux.
Avec `matplotlib`, pour tracer une droite, vous pouvez très bien faire `plt.plot([x1,x2], [y1,y2])`.
4. Bien sûr, `regrelinEntre` va se contenter de créer les sous-tableaux et appeler `regrelin`.
5. Pas de soucis pour peu que le fichier se trouve dans le répertoire de travail. Sinon, il faudra prévenir spyder ou pyzo pour lui dire où aller chercher les fichiers.

4 Solutions...

4.1 Blondes

Allez, courage... vous avez toutes les bases « théoriques », yapluka pratiquer !

4.2 Châtain clair...

.1 Courbe de Blanc-Mange

```

1 def B(n,x):
2     b = 0
3     for k in range(0, n+1):
4         u = (2**k)*x
5         b = b + abs((u-floor(u+0.5))/(2**k))
6     return b
7
8 n = 1000
9 X = []
10 Y = []
11 x = 0
12 while x <= 1 :
13     X.append(x)
14     Y.append(B(n,x))
15     x = x + 0.001
16
17 plt.plot(X,Y)
18 plt.show()
```

J'ai calculé $(2 * k) * x$ à part pour me simplifier la vie avec les parenthèses mais ce n'est pas nécessaire.

4.3 Triangle de Pascal

```

1 def printFormat(liste):
2     chaine= ""
3     for n in liste :
4         chaine = chaine + "{:4d}".format(n)
5     print (chaine)
6
7 def ligneSuivante(ligne) :
8     ligne2 = [1]
9     for i in range(len(ligne)-1):
10         ligne2.append(ligne[i]+ligne[i+1])
11     ligne2.append(1)
12     return ligne2
13
14
15 ligne = [1]
16 for i in range(8) :
17     printFormat(ligne)
18     ligne = ligneSuivante(ligne)
```

J'ai créé une fonction printFormat pour « clarifier » un peu le code, même si on pouvait s'en passer.

On retrouve, ici, la principale difficulté des listes... la gestion correcte des indices !

4.4 Réactions successives

```

1  from math import exp
2  #question 1
3
4  A0 = 1
5
6  def A(k1,k2,t):
7      return A0*exp(-k1*t)
8
9  def B(k1,k2,t):
10     return (A0*k1/(k2-k1))*(exp(-k1*t)-exp(-k2*t))
11
12 def C(k1,k2,t):
13     return A0 - A(k1,k2,t)-B(k1,k2,t)
14
15 def prepare(k1,k2,tmax,dt):
16     tT=[]
17     tA=[]
18     tB=[]
19     tC=[]
20     t = 0
21     while t < tmax :
22         tT.append(t)
23         tA.append(A(k1,k2,t))
24         tB.append(B(k1,k2,t))
25         tC.append(C(k1,k2,t))
26         t = t + dt
27     return tT, tA, tB, tC
28
29 T, cA, cB, cC = prepare(1,2,5,0.01)
30
31 plt.plot(T,cA, 'r')
32 plt.plot(T,cB, 'g')
33 plt.plot(T,cC, 'b')
34 plt.show()
35
36 #question 2
37
38 def rechercheMax(X):
39     maxi = X[0]
40     indice = 0
41     for i in range(len(X)):
42         if X[i] > maxi :
43             maxi = X[i]
44             indice = i
45     return indice, maxi
46
47 imax, bmax = rechercheMax(cB)
48 print("Bmax = ", bmax, " pour t = ", T[imax])
49
50 # question 3

```



```

51
52 LogR = []
53 Bmax = []
54 logr = -2
55 while logr <= 2 :
56     if abs(logr) > 0.1 : # problème si k1 = k2 dans B!
57         r = 10**logr
58         T, cA, cB, cC = prepare(1,r,10,0.01)
59         imax, bmax = rechercheMax(cB)
60         LogR.append(logr)
61         Bmax.append(bmax)
62     logr = logr + 0.1
63
64 plt.plot(LogR, Bmax)
65 plt.show()
66
67 # question 4
68
69 i = 0
70 trouve = False
71 while not trouve and i < len(Bmax):
72     if Bmax[i] <= 0.05 :
73         trouve = True
74         rAEQS = 10**LogR[i]
75     i = i + 1
76 if trouve :
77     print ("AEQS pour k2 > " ,round(rAEQS), " k1")

```

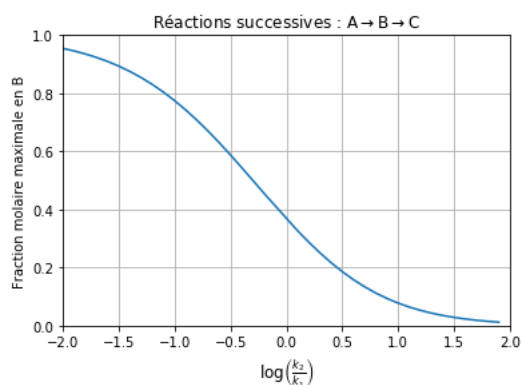
1. Le codage des différentes fonctions ne devrait pas poser trop de difficultés. Pour pouvoir facilement tester différents cas de figure et n'avoir à changer qu'une seule fois les valeurs de k_1 et k_2 , j'ai rajouté la fonction `prepare` qui génère les 4 tableaux nécessaires. On aurait pu mettre k_1 et k_2 comme variable globale s'il n'y avait pas eu la question 3! On pourrait enrichir le graphe!
2. On retrouve le code de recherche d'un extremum avec l'indice de cet extremum.
3. Pour chaque valeur de r , il faut générer le tableau de valeur $B(t)$. On aurait très bien pu écrire une fonction qui ne calcul que B , vue la rapidité des calculs sur les ordinateurs, autant conserver la fonction génère du 1. Attention à bien stocker toutes les variables, même celle qui ne servent à rien! C'est normal, au début, d'être un peu perdu par tous ces appels de fonction, j'en dirai quelques mots dans le chapitre suivant. Bref, en évitant un accident de parcours pour r très proche de 1, on a une jolie courbe.
4. Ici, on cherche une valeur seuil. On aurait très bien pu le faire en même temps que la question 3. J'ai préféré reprendre le code correspondant. Il faut faire attention à s'arrêter dès que B_{max} devient plus petit qu'une valeur déterminée et sortir de la boucle `while`.

Pour les latexiens, la bibliothèque `matplotlib` permet un enrichissement des graphes :

```

1 plt.xlabel(r"$\log\left(\frac{k_2}{k_1}\right)$", fontsize=12)
2 plt.ylabel("Fraction molaire maximale en B")
3 plt.title(r"Réactions successives :
    A$\rightarrow$B$\rightarrow$C")

```



4.5 Régression linéaire

```

1  def moyenne(X):
2      m = 0
3      for x in X :
4          m = m + x
5      return m/len(X)
6
7  def ecart_type(X):
8      m = moyenne(X)
9      u = 0
10     for x in X :
11         u = u + (x-m)**2
12     return sqrt(u/len(X))
13
14  def cov(X,Y):
15     mx = moyenne(X)
16     my = moyenne(Y)
17     c = 0
18     for i in range(len(X)):
19         c = c + (X[i]-mx)*(Y[i]-my)
20     return c/len(X)
21
22  def regrelin(X,Y):
23     pente = cov(X,Y)/ecart_type(X)**2
24     ordonnee = moyenne(Y)-pente*moyenne(X)
25     coeff = cov(X,Y)/(ecart_type(X)*ecart_type(Y))
26     return pente, ordonnee, coeff
27
28  def regrelinEntre(X,Y,xmin,xmax):
29     nX = []
30     nY = []
31     for i in range(len(X)):
32         if X[i] >= xmin and X[i] <= xmax :
33             nX.append(X[i])
34             nY.append(Y[i])
35     return regrelin(nX, nY)
36
37  # test
38  X=[1,2,3,4,5]
39  Y=[3.1, 4.9,7,8.9,11.2]
40

```

```

41 a,b,r = regrelin(X,Y)
42 print(" a = ", a, " b = ", b, "r^2 = ", r**2)
43
44 # détermination d'une énergie d'activation
45 theta=[35,40,45,50]
46 k=[0.188,0.257,0.354,0.477]
47
48 unSurT = [1/(t+273) for t in theta]
49 lnk = [log(x) for x in k]
50
51 a,b,r = regrelin(unSurT, lnk)
52
53 print("Energie d'activation : ", round(- 8.314*a/1000 ), "
      kJ/mol")
54 print("r^2 = ", r**2)
55
56 plt.plot(unSurT, lnk,'o')
57 plt.plot([unSurT[0], unSurT[len(unSurT)-1]], [a*unSurT[0]+b,
      a*unSurT[len(unSurT)-1]+b])
58 plt.show()

```

1. Les paramètres des fonctions sont muets, `moyenn(X)` peut traiter n'importe quel nom de tableau, qu'il s'appelle *X*, *Y* ou *schtrumph_grincheux*.
On stocke, bien sûr, dans des variables les moyennes calculées dans les fonction `ecart_type` et `cov`.
2. Le test proposé devrait donner les bons résultats!
3. On aurait, bien sûr, pu créer les listes *unSurT* et *lnk* avec des `append`. Quoiqu'il en soit, c'est peut-être plus rapide que d'expliquer les fonctions de tableau dans Graph2D!
Qui a oublié le 273?

Amis de l'oral de Centrale, on pourrait alors, pour un exercice d'oral, imaginer quelque chose comme ça à la place de Graph2D!

```

1 from math import *
2 import matplotlib.pyplot as plt
3 from utilitaire import *
4
5 """ Routines utilitaires
6
7 Vous disposez dans la bibliothèque utilitaire de deux
      fonctions permettant d'effectuer une régression linéaire
8
9 a,b,r = regrelin(X,Y)
10 a,b,r = regrelinEntre(X,Y,xmin,xmax)
11
12 X, Y : tableaux des abscisses et des ordonnées (de même
      dimension)
13 a : pente de la droite de régression
14 b : ordonnée à l'origine de la droite de régression
15 r : coefficient de corrélation
16

```

```

17  la seconde fonction permet de n'effectuer la régression
    qu'entre les abscisses xmin et xmax
18
19  """
20
21  """ Données de l'exercice """
22  theta=[35,40,45,50] # température (en degrés Celcius)
23  k=[0.188,0.257,0.354,0.477] # constante de vitesse
    (mol-1.L.s-1)
24
25  """ Partie à modifier par le candidat """
26
27
28
29  #plt.plot(X, Y, 'o') # X et Y tableaux de même dimension, 'o'
    pour ne représenter que les points expérimentaux
30  plt.show()

```

5

Annexe : bibliothèque matplotlib

Il ne s'agit pas, ici, de résumer toutes les possibilités de la bibliothèque matplotlib. On pourra se reporter à la page : <http://matplotlib.org> puis aux onglets « exemples » ou « docs » pour exploiter au mieux cette bibliothèque.

Cette bibliothèque permet le tracé de courbes. Dans la version minimale, il suffit de remplir deux tableaux de valeurs (de même taille) et d'appeler le tracé.

```

1  import matplotlib.pyplot as plt
2  X = []
3  Y = []
4  x = 0
5  while x <=5 :
6      X.append(x)
7      Y.append(x/(1+x))
8      x = x + 0.1
9
10 plt.plot(X, Y)

```

L'exemple ci-dessous permet d'enrichir le graphique :

- On ajoute une étiquette sur chaque axe, un titre et la légende des courbes.
- On précise, entre guillemet, que pour la courbe Y en fonction de X , on ne représente que les points tabulés sous forme de ronds rouges et que la courbe Z est représentée en pointillé bleu.
- On a spécifié (lignes 18 et 19) les intervalles de tracé sur chacun des axes.
- On ajoute, ligne 20, du texte aux abscisses et ordonnées spécifiées en noir et avec une fonte de taille 20.
- La ligne 21 permet de préciser les étiquettes représentées sur l'axe y.
- On ajoute une grille (ligne 22).

```

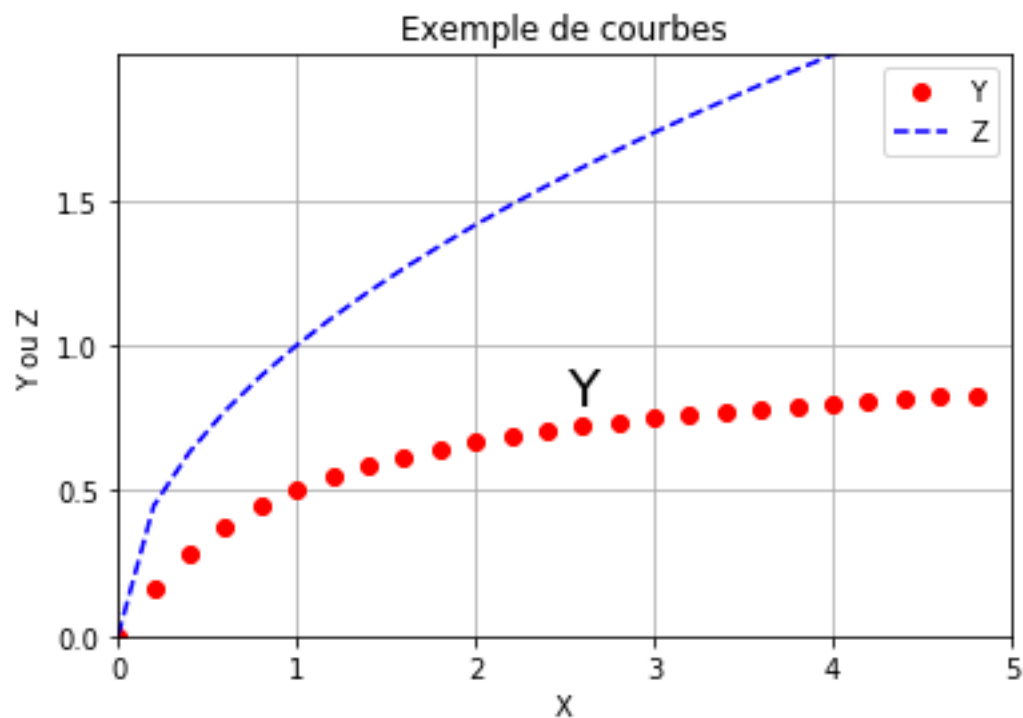
1  import matplotlib.pyplot as plt
2  from math import sqrt

```

```

3 X = []
4 Y = []
5 Z = []
6 x = 0
7 while x <=5 :
8     X.append(x)
9     Y.append(x/(1+x))
10    Z.append(sqrt(x))
11    x = x + 0.2
12
13 plt.plot(X, Y, "ro", label="Y")
14 plt.plot(X, Z, "b--", label="Z")
15 plt.xlabel("X")
16 plt.ylabel("Y ou Z")
17 plt.title("Exemple de courbes")
18 plt.xlim(0,5)
19 plt.ylim(0,2)
20 plt.text(2.5, 0.8, "Y", {'color': 'k', 'fontsize': 20})
21 plt.yticks([i/2 for i in range(0,4)])
22 plt.grid()
23 plt.legend()
24 plt.show()

```



En s'aidant de l'aide en ligne, on peut tracer un graphe avec deux ordonnées.

Les tableaux V , PH et G contiennent, respectivement, volume, pH et conductance mesurés lors d'un titrage.

Le code ne s'improvise pas mais, vive le copier-coller !

```

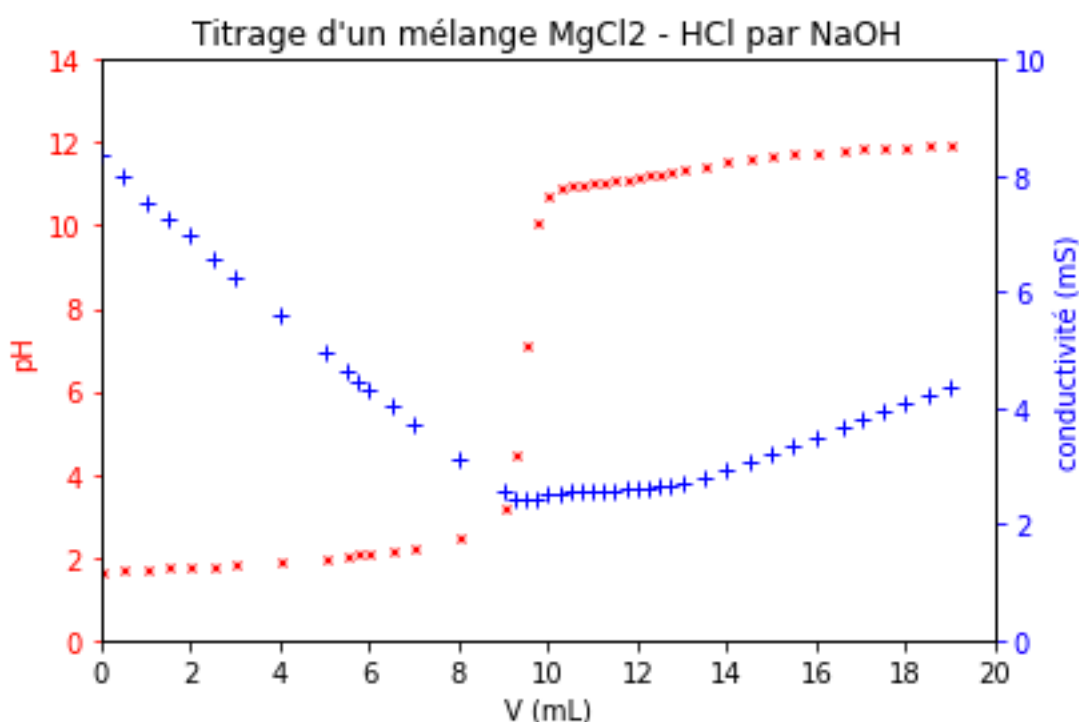
1 fig, ax1 = plt.subplots()
2 ax1.plot(V, PH, "rx", markersize=3)
3 ax1.set_xlabel("V (mL)")
4 ax1.set_ylabel("pH", color="r")

```

```

5 ax1.tick_params("y", colors="r")
6 ax1.set_ylim(0,14)
7
8 ax2 = ax1.twinx()
9 ax2.plot(V, G, "b+")
10 ax2.set_ylabel("conductivité (mS)", color="b")
11 ax2.tick_params("y", colors="b")
12 ax2.set_ylim(0,10)
13 ax2.set_xlim(0,20)
14 ax2.set_xticks([i for i in range(0,22,2)])
15
16 plt.title(r"$Titration d'un mélange MgCl2 - HCl par NaOH")
17 plt.show()

```



On peut même réaliser des animations... en fait, c'est fabuleux tout ce qu'on peut faire avec matplotlib; et encore, on n'est « que » chimiste!

Les latexiens peuvent même écrire en latex²... en écrivant les chaînes de caractères attendues sous la forme `r"expression latex"`. Par exemple, avec `plt.title(r"Titration d'un mélange MgCl2 - HCl par NaOH")` on obtient MgCl_2 ; on peut bien sûr, avoir également des lettres grecques...

2. Sans savoir comment, ça marche chez moi mais il y a peut-être des liens à établir?